Notes on reading

I handed out a paper presented at the conference I was at last week: *The Architecture of Montana: An Open and Extensible Programming Environment with an Incremental C++ Compiler*, by Michael Karasick of IBM. This is compulsory reading: the reason is that it's important to see some real (non-toy) examples of architectural decisions, and this paper contains some good discussion of them. You will need to read it fairly carefully to understand what's going on architecturally, and I expect you may find it quite hard going. It's worth perservering but remember that the point is to be able to understand practical architectural decisions not to be able to add a tool to this environment! Watch out for demonstrations:

- that architectural decisions are not just about high level structure: they include all kinds of conventions about the way that components work together. The architecture is what ensures that components plug together.
- that there are often interesting trade-offs, e.g. between performance and flexibility
- that the type of organisation, e.g. its staffing turnover, can have an effect on what the best architecture is

The fact that the paper frequently refers to "observers" was a good excuse to cover this common pattern. (It also talks about Visitors: you do not need to know this pattern, but broadly it is a neat way to avoid having type case statements (e.g. if foo has class Foo then ... else if it has class Bar then...))

Observer

This pattern is known either as Observer or as Publisher-Subscriber, and it's about ways of controlling dependencies between things that have to be kept up to date.

One common problem

It often happens that when one object changes state, one or more other objects need to know about it so that their states remain consistent.

E.g. one or more elemements of a user interface need to show the correct version of some data item, part of the state of an object (say aCustomer) inside the system.

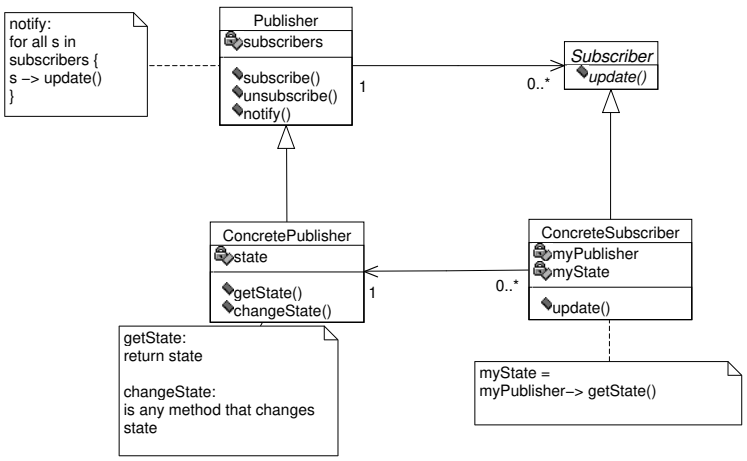
We could introduce explicit dependencies, and have the changing object notify its dependants by sending them messages: but this is inflexible and prevents reuse (e.g. of aCustomer). User interfaces are very prone to change, so internal objects shouldn't know what the user interface is like.

We could have the dependants poll the changing object to see when it changes: but this is very inefficient.

May well want to add or remove dependants at run time.

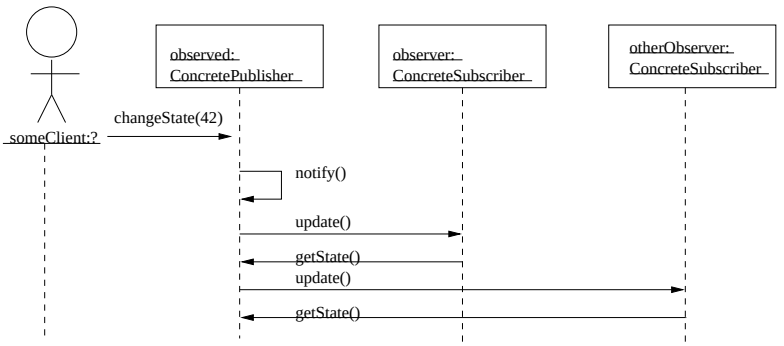
Slide 16

Class diagram



Slide 17

Publisher/subscriber interaction



This sequence diagram does not show activations or return arrows.

Exercise (highly recommended): check that you understand the interaction by adding them in.

Slide 18

We haven't given this pattern in pattern format, but here's a quick reminder of what that looks like:

Elements of a pattern

A pattern catalogue entry normally includes these elements:

- Name (e.g. Publisher-Subscriber)
- Aliases (e.g. Observer, Dependants)
- Context (in what circumstances can the problem arise?)
- Problem (see slide above)
- Solution (see slides above: normally a mixture of text and models)
- Consequences (good and bad things about what happens if you use the pattern.)

The elements aren't always named in this way. GoF lump context and problems together, Buschmann et al don't have a separate consequence section.

It may contain more, e.g. variations, CRC cards, example code.

Slide 19